N91-22792

DESIGN OF AN INTELLIGENT INFORMATION SYSTEM FOR IN-FLIGHT EMERGENCY ASSISTANCE

Stefan Feyock
Stamos Karamouzis

Computer Science Department
College of William & Mary
Williamsburg, VA 23185

Abstract

The present research has as its goal the development of AI tools to help flight crews cope with in-flight malfunctions. The relevant tasks in such situations include diagnosis, prognosis and recovery plan generation. Investigation of the information requirements of these tasks has shown that the determination of paths figures largely: what components or systems are connected to what others, how they are connected, whether connections satisfying certain criteria exist, and a number of related queries. The formulation of such queries frequently requires capabilities of the second-order predicate calculus. We describe an information system that features second-order logic capabilities, and is oriented toward efficient formulation and execution of such queries.

1. Introduction

The research described in this report was performed in conjunction with the Intelligent Cockpit Aids (ICAT) project conducted by the Vehicle Operations Branch at NASA Langley Research Center. The goal of this project is to develop artificial intelligence (AI) techniques and systems to assist flight crews in the performance of their tasks. Such assistance can become particularly crucial when malfunctions occur; a significant portion of the project is accordingly devoted to the development of tools that will help flight crews cope with in-flight faults.

The design possibilities for such software tools range from passive systems to be used as information resource by the flight crew, through systems that autonomously determine and suggest appropriate actions, to software that takes complete control of the aircraft in case of emergency*. We begin by considering the low-autonomy end of this spectrum, with systems that confine themselves to producing answers to queries posed by the flight crew. Once the functions of such systems are understood, we can consider what aspects of the system's *user* can be automated.

The questions that arise are:

1. what type of information is likely to be required in case of in-flight malfunctions, and

* The set of pilots who consider the latter class of systems to be a good idea is essentially null; nonetheless, certain recently developed aircraft types exhibit a disquieting degree of autonomy in deciding how to reconfigure themselves after a malfunction.

2. how is it to be produced?

As regards item (1), it is clear that the most critical issues to be determined are:

1. diagnosis: what is the nature of the fault?

2. prognosis: how will this fault affect the subsequent operation of the aircraft, in terms of flight characteristics as well as fault propagation to additional components?

3. recovery planning: what is the appropriate response to the malfunction?

Ongoing research by members of the ICAT group has resulted in a number of sophisticated AI tools appropriate to the task of producing in-flight diagnoses [1,7]. The research described in this paper has concentrated on the production of *prognoses*, given a diagnosis produced by such diagnostic systems. In the course of our work we have come to the following conclusions:

1. traditional information (i.e. database) systems technology is inadequate for the task of prognosis. For example, we have found that the underlying database system must be able to respond to queries of the type "is there an electrical path between components A and B? A hydraulic connection? Any sort of connection? Such queries involve quantification over paths and relations, and thus belong to the realm of the second-order predicate calculus, beyond the capabilities of traditional database systems.

2. Standard rule-based expert systems also fail to meet the requirements of the task. The rules within such systems are to a large extent distillations of experts' responses to familiar problem situations. Malfunctions by their very nature create chaotic conditions rife with unforeseen consequences that may interact in unexpected ways. The brittleness of rule-based systems in the face of such situations is well-known; a deeper kind of reasoning is required to generate adequate responses.

Our approach to these problems has been to embed a variety of models in our information system to allow deep reasoning to take place, and to develop a database system capable of the second-order operations that occur frequently in the course of such reasoning. Fig. 1 depicts the overall organization of the resulting system.

As can be seen, the model-based information system (MBIS) consists of a number of submodels of a quite diverse nature, coordinated by an entity which may be an in-the-loop human, or may be an "automated flight engineer" that assumes the human user's functions to the extent feasible. The LIMAP utility is the second-order database discussed above, while the semantic net serves as a central information resource helping to tie the submodels of MBIS together. In this paper we describe the LIMAP submodule of the MBIS system, and how it interacts with the other components to fulfill its role.

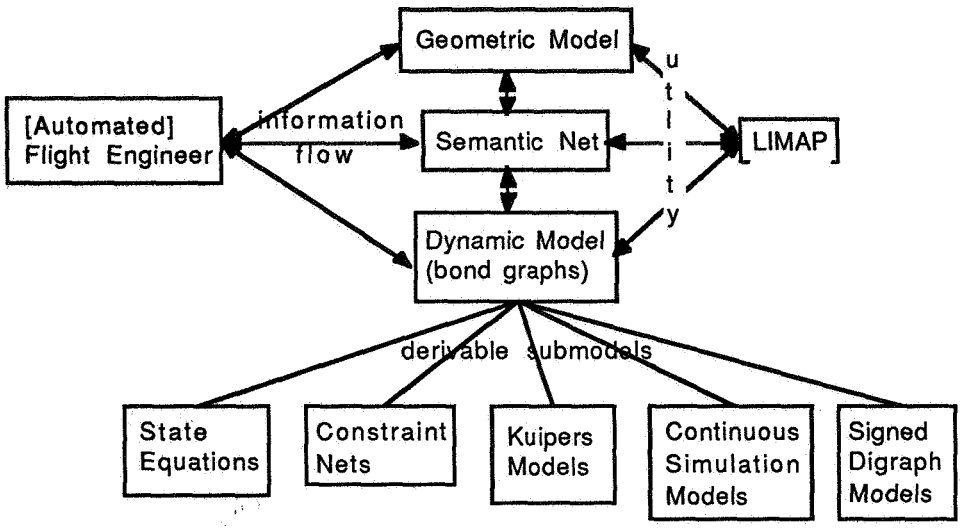

Fig. 1 The Model-Based Information System

2. Second-Order System: Rationale

We will introduce LIMAP by describing a typical situation in which the need for a second-order database system arises. As indicated above, ICAT researchers have developed a number of diagnostic tools to assist the flight crew with the task of determining the cause of in-flight malfunctions. One of the most important of these is the DRAPHYS system [1], a model-based reasoner that determines which components' malfunction best accounts for the observed symptoms. DRAPHYS uses a digraph model of an aircraft system, with nodes representing primitive components*, and the arrows connecting nodes representing functional and physical dependencies. Component B is said to be *functionally dependent* on component A if the proper functioning of B depends on the proper functioning of A. Component B is *physically dependent* on component A if damage to A can propagate through space to component B. For example, the control surfaces of an aircraft are functionally dependent on the hydraulic system, since they will cease operating if the latter fails. On the other hand, if a hydraulic line can be severed by a disintegrating turbine, the line is physically dependent on the turbine. While functional dependencies can generally be determined by considering causal relationships in the physical system, physical propagation is typically the result of leakage of some substance or form of energy (e.g. as in an explosion), and is inherently unpredictable. Reference [5] contains a discussion of model-based reasoning about physical fault propagation.

We will illustrate the concepts involved by means of an example: a jet engine. Fig. 2 show a schematic of a dual-fan jet engine, while Fig. 3 gives the functional dependency graph for this engine. (we ignore physical dependencies for the sake of simplicity)

---

* A component is deemed to be *primitive* if it is considered to be atomic, i.e. to have no subcomponents, with respect to the model. This property clearly depends on the granularity of the model; thus, an entire engine could be treated as primitive if we are content with the diagnosis "engine malfunction" rather than, say, "compressor stall".

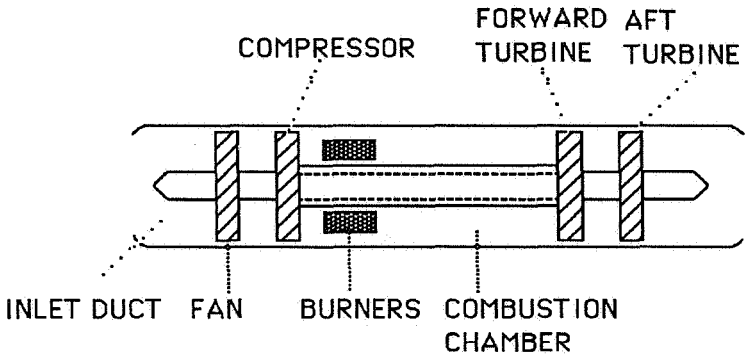

Fig. 2 Jet Engine: Schema

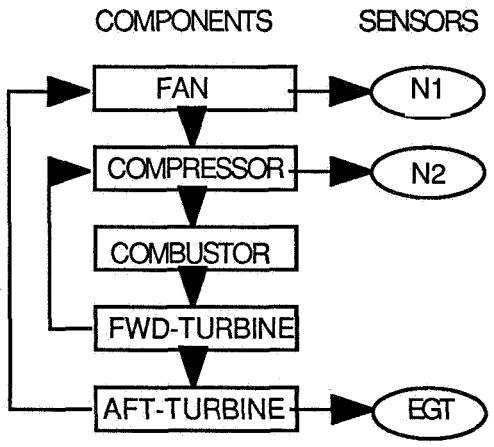

Fig. 3 Functional Dependency Model of Jet Engine

When a malfunction occurs, the DRAPHYS model-based diagnostic system extracts information from this model by means of the following procedure:

Suspect every possible component*;
Use model to determine consequences of each failure;
Eliminate suspects inconsistent with model predictions;

More precisely:

```
for each C in SET-OF-PRIMITIVE COMPONENTS do
        if "C has failed" is a valid hypothesis
        then add C to SET-OF-VALID-HYPOTHESES;
end for;
```

* Heuristics are used to prune the set of "possible" components.

With luck, SET-OF-VALID-HYPOTHESES will contain only one element. If it contains more, DRAPHYS waits for more symptoms to develop and disambiguate the diagnosis.

We must specify how it is determined that "C has failed" is a valid hypothesis (possible diagnosis). Here is an informal description of how this is done:

> Primitive component C is a valid hypothesis iff
> there is a POSSIBLE PROPAGATION PATH from C
> to every symptomatic sensor

A path is a POSSIBLE PROPAGATION PATH iff every instrumented component on the path has at least one symptomatic sensor

Here is a predicate calculus formulation of this stipulation:

```
valid-hypothesis(component)  <=>

is-primitive(component)  &
(A s) {is-sensor(s) & is-symptomatic(s) =>
(E p) [(path(p, /*from*/ component, /*to*/ s) &
(A n)[is-node(n, /*on path*/ p) => not is-ok(n) or is-unknown(n)]]}

is-ok(n) <=> (A s)[is-sensor(s) & instruments(s,n) =>
      not is-symptomatic(s)]

is-unknown(n) <=>
      not (E s)[is-sensor(s) & instruments(s,n)]
```

/* instruments(s,n) is true if s is a sensor measuring some attribute of n */

This definition contains the construct "does there exist a path from x to y such that every node on that graph fulfills certain constraints?". The node constraint in this case is that every instrumented node on the path must be symptomatic. The wff states that a primitive component C is a valid hypothesis iff there is such a path from C to every symptomatic sensor. Such quantifications are difficult to express in the first-order predicate calculus; if each edge of a graph can denote a distinct arbitrary relation, a second-order formulation is required, since it must contain (among other things), a passage such as this: "...for all nodes m and n on path p there exists relation R such that R(m,n)...".

Queries of the form "is there a relation R such that nodes a and b are in relation R? "is there a path from x to y? a path fulfilling constraint C? where can I go from x? how can I get to x?" arise frequently both in AI and out of it. Such queries, which involve quantification over relations, correspond to statements in the second order PC. The question is: how can such queries be implemented efficiently? In the following section we describe the structure and implementation of an information system appropriate to the requirements we have outlined.

3. LIMAP

The second-order operations that have been discussed raise the question of whether existing languages, particularly Prolog [3], already fulfill all the

stated requirements. Some Prolog implementations do allow predicates to be variables, permitting at least existential quantification. Furthermore, it is straightforward to construct Prolog procedures that determine the possible paths between nodes in an arbitrary network, as well as allowing the user to stipulate constraints on these paths, to browse through them, and to perform a variety of similar operations. The problem is that such capabilities can easily become prohibitively inefficient. This is the motivation for the creation of LIMAP, which offers a number of Prolog-like capabilities, but whose implementation is oriented toward storage and execution efficiency.

Most AI search/representation techniques are oriented toward a potentially infinite domain of objects and arbitrary relations among them. Experience has shown that in practice much of what needs to be represented in AI can be expressed using a finite domain and unary or binary predicates. Even in cases where large relations are involved, they are almost invariably sparse as well. A major subassembly of an aircraft, for example, may have thousands of subcomponents, but the relations of interest among them, such as IS-A, PART-OF, functional dependency, etc., each involve only a small proportion of the full set of components.

As regards the arity of predicates, the most widespread and useful representations of AI are semantic nets, frames, schemas, inheritance hierarchies, and related constructs. All of these can be expressed by means of predicates of at most two arguments*.

3.1. Representations of Predicates

Since most of the relations of interest will be sparse and have no more than two arguments, specialized techniques for representing and manipulating such relations may be used. Unary predicates over a finite fixed domain are well represented by bit strips (boolean vectors), while adjacency matrices (AM's; see Appendix) are appropriate for binary predicates. A representation based on (optionally sparse) vectors and adjacency matrices accordingly forms the basis of the LIMAP implementation.

By way of example, Fig. 4 depicts the adjacency matrix representing the jet engine functional dependency predicate** FD(x,y) of Fig. 3 over the domain D = {fan, compressor, combustor, fwd-turbine, aft-turbine, N1-sensor, N2-sensor, EGT-sensor}. By definition of adjacency matrices, a "1" in row i, column j denotes an arrow from node i to node j.

Boolean vectors are equally useful for representing unary predicates. The vector IS-SENSOR = <0, 0, 0, 0, 0, 1, 1, 1>, for example, represents the IS-SENSOR predicate; a "1" in position i denotes the fact that element i is a sensor.

---

* There are, of course, well-known techniques for expressing any n-ary (n > 2) predicate in terms of binary predicates. Most knowledge can be expressed using unary and binary predicates without performing such a decomposition.

** The notation is somewhat misleading: FD(x,y) iff x *functionally determines* y, i.e. iff y is functionally dependent on x.

	1	2	3	4	5	6	7	8
1. fan		1				1		
2 compressor			1				1	
3 combustor				1				
4 fwd-turbine		1			1			
5 aft-turbine	1							1
6 N1 sensor								
7 N2 sensor								
8 EGT sensor								

Fig. 4

Given an AM representation, many useful second-order operations can be expressed concisely and efficiently:

(EXISTS X) X(a,b)?
(FORALL X) X(a,b)?
(EXISTS P) P a path from node a to node b?

Such queries correspond to finding parents, siblings, descendants, routes between nodes, etc. For example, we can obtain the set of instrumented components simply by performing the boolean matrix multiplication FD x IS-SENSOR^T. Similarly, the question of whether a path exists between, say, the fan and the EGT sensor is trivially answered by noting whether FD*[1,8] contains a 1.

It is frequently necessary, however, to determine not only the existence of a path, but the path itself, as ordered set of nodes. In addition, if the problem representation requires a large number of predicates over D or D x D, maintaining a separate AM for each predicate becomes unwieldy.

A straightforward extension of the adjacency matrix representation allows digraphs with labeled edges to be represented. Under this extension, the elements of the AM are sets of labels, rather than 0 or 1. Element ij contains label P iff the semantic net contains an arrow from d_i to d_j representing (labeled with) predicate P; the empty set denotes the absence of an edge. Such extended AMs are termed *symbolic adjacency matrices* (SAMs).

It is worth noting that SAMs provide a mechanism for implementing second-order queries. The wff (EXISTS X) X(a,b), for example, can be decided simply by determining if row a, column b is the empty set.

3.2. PSAMs

A straightforward extension of Warshall's Algorithm to symbolic adjacency matrices allows efficient computation of a *path symbolic adjacency matrix* (PSAM; see Appendix), a matrix whose ij entry contains the set of all paths from node i to node j. It is this capability that makes quantification over paths feasible.

4. The LIMAP language

The current version of LIMAP is an experimental test bed for determining what facilities are required, and how to implement them most efficiently. In this section we describe the language features that have been implemented to

date, and plans for future.development. The prototype version of LIMAP is implemented in Common LISP.[8]. A C implementation, oriented toward optimizing the bit level representations and operations on which the efficiency of LIMAP depends, is under way.

4.1. The LIMAP DDL/DML

Every language has an underlying implementation model. As we have seen, the LIMAP implementation model is based on a representation that employs boolean and symbolic vectors and adjacency matrices to represent unary and binary predicates, as well as an efficient transitive closure computation capability that allows boolean or symbolic path matrices to be computed and manipulated.

As is the case for an ordinary first-order database system, LIMAP capabilities are invoked via a language interface that consists of two parts. One is the data definition language (DDL) for specifying both the data the system is to contain as well as "metadata", i.e. information about the structure and constraints that govern the data contained in the system. The other is the data manipulation language (DML), the subset of the language concerned with the specification of queries and updates on the the data. We will categorize.the LIMAP functions accordingly. A brief summary of the LIMAP DDL and DML follow; [4] contains a complete listing.

4.1.1. DDL operations

The basic DDL operations are

DEFREL <name> <specification> <type> <representation>
<specification> ::= (<number>) or (<number> <number>)
<type> ::= boolean | symbolic
<representation> ::= sparse | dense

and

DELREL <name>

to define, respectively delete, a relation. DEFREL creates a new array according to the values of the parameters, and binds this array to <name>. <specification> stipulates whether the array will be a vector or matrix, as well as the index range(s). <type> specifies whether the declared relation will be represented by a boolean or symbolic array, whereas <representation> allows the user to choose a sparse or dense array representation. Defaults are provided for the SPECS and REP parameters. The attributes of an array, as well as a pointer to the data structure representing the array elements, are inserted into an internal symbol tables. As might be expected, the effect of DELREL <name> is to unbind <name>, effectively deleting the array.

4.1.2. DML operations

The major DML operations are

STORE	relname	value [row] column	Store value
RETRIEVE	relname	[row] column	Retrieve contents
TCLOSE	relname		Transitive closure
PATHS	relname	row column	All paths
MULT	relname	relname	Multiply
TRANSPOSE	relname	relname	Transpose

STORE and RETRIEVE perform the indicated operation on the specified array position, in accordance with the array's type and representation, while MULT and TRANSPOSE typify a variety of standard matrix operations made available by LIMAP. Except in DEFREL it is transparent to the user whether the array representation is sparse or dense. This transparency extends to the other attributes of the array wherever possible.

4.1.3. Path Operations

The TCLOSE and PATHS operations form the core of LIMAP's path manipulation capability. TCLOSE computes the transitive closure or the PSAM of the indicated array, depending on its type. A pointer to the resulting closure array is stored in the symbol table for relname, and may henceforth be accessed by queries referencing paths. In particular, PATHS[i j] retrieves the set of all paths from node i to node j, enabling quantification over paths. The LIMAP implementation of DRAPHYS, shown below, contains an example of this capability.

4.2. Control structures

The distinction between procedural and non-procedural predicate calculus specifications blurs if the underlying domain is finite, since the FORALL and EXISTS quantifiers map in an obvious way to loops ranging over the domain elements. It has been our goal to give the LIMAP DML as non-procedural a character as possible. In particular, LIMAP notation is an adaptation of the (function-less) predicate calculus, with extensions to allow data retrieval in addition to data specification*, .Perhaps surprisingly, we have found that minimal modifications of the control macros described in [2] were suitable for the task of expressing the required quantifications. Here is a summary of the general form of the control structure implemented by these macros:

(FOR ((<variable$_1$> :IN <set$_1$>)

(<variable$_n$> :IN <set$_n$>))
[:WHEN <when-expression>]
<FOR-keyword> <expression$_1$> ... <expression$_n$>)

The construct (<variable$_i$> :IN <set$_i$>) causes the variable to iterate over the elements of the set, which may be specified as a list, a vector, or a matrix row or column. Unless a false when-expression is present, the FOR-body is evaluated and a result is produced as governed by the FOR-keyword. Iteration then proceeds to the next set of variable values.

FOR-keywords

:ALWAYS	true if all the values of body are true
:FILTER	produce a list of the non-NIL values of body
:FIRST	produce the first non-NIL value of body
:SAVE	produce a list of all values of body

While the description of these constructs is procedural in form, the effect when programming in this notation is that of writing FORALLs and EXISTs, with the proviso that any variable values that are found to "EXIST" are col-

* For example, a "yes" answer to (EXISTS X)(FORALL Y)P(X,Y) is insufficient; the actual X-value must be retrieved.

lected in accordance with the FOR-keyword and returned as value. The following section contains an example application of LIMAP: a LIMAP specification of DRAPHYS.

5. DRAPHYS in LIMAP

The operation of DRAPHYS has been described previously, both in informal terms and by means of predicate calculus wffs. Here is a LIMAP version; since the notation bears strong analogies to the predicate calculus specification, we present it without further explanation.

```
; DRAPHYS in LIMAP

; The list COMPONENTS contains all engine components, including sensors
;
(defun determine-hypotheses (components symptomatic-sensors)
; components =def set of all components to be considered as hypotheses
(for (c :in components) :when (is-valid-hypothesis c) :filter c)
)

(defun is-valid-hypothesis (c symptomatic-sensors)
(for (s :in symptomatic-sensors) :always (exists-bad-path c s) )
)

(defun exists-bad-path (c s)
 (for (p :in (paths 'engine c s) ) ; paths from c to s
       :first (for (c :in p) :always (not-known-ok c) )
))

(defun not-known-ok (c)
  (or (null (instrumentation c)) (symptomatic c))
) ; symptomatic is a boolean vector

(defun instrumentation (c)  ;  returns list of sensors associated with c
(for (s :in components)
:when (and (is-sensor s) (retrieve 'engine c s)) :save s)
)
```

6. Conclusion

We have described a programming system oriented toward efficient information manipulation over fixed finite domains, and quantification over paths and predicates. The initial motivation for the creation of such a system was the fact that the need for such operations arose frequently in the diagnosis/prognosis generation problem domain. Since then it has become apparent that the facilities provided are useful and applicable over a much wider range of problems, both within and outside of AI.

LIMAP's predicate-oriented DDL and DML are reminiscent of another predicate-oriented language: Prolog. There is, however, an important omission: Prolog contains a built-in inference engine for processing rules, while LIMAP does not. As it happens, since SAM entries can be arbitrary s-expressions, rules are easily added to LIMAP. This is an artifact of the fact that the current implementation language is LISP, and does not generalize to other (planned) implementation vehicles such as C. Addition of a rule capability and inference

engine forms a major area of current research, as does optimizing implementation efficiency.

Our experience to date has shown that LIMAP is applicable to a wide range of problems. While LIMAP, if abused, is as capable of inefficient operation as any other misused programming system, we have found that for every problem yet attempted there has existed a LIMAP formulation that was concise, comprehensible, and for which LIMAP's facilities constituted a highly efficient problem representation.

APPENDIX

We present a brief review of graph-theoretical terminology occurring in the text; see [6] for a detailed discussion.

Digraphs

A directed graph (digraph) is 2-tuple <N,E>, where N is a finite set of nodes, and E a finite set of edges. An edge is a member <a,b> of NxN. A labeled digraph is a 3-tuple< N,E,L>, where N is as before, L is a finite set of labels, and E is a finite set of labeled edges, with labels in L. A labeled edge (with label in L) <a,l,b> is a member of NxLxN.

It is easy to see that digraphs are a graphic representation of binary predicates over finite domains If P(x,y) is a predicate over domain DxD, then digraph G = <N,E> represents P if P(a,b) iff <a,b> in E.

Whereas an unlabeled digraph can represent a single predicate, labeled digraphs whose label set is a set of predicate names can represent multiple binary predicates over the same domain DxD simply by letting edge <a,p,b> denote the fact that predicate p(a,b) is true; the absence of such an edge denotes that p(a,b) is false. Extending the notation, we allow edges to be labeled with *sets* of predicate names; an edge $<a,\{p_1,...,p_n\},b>$ is an abbreviation for the set of edges $<a,p_1,b>$, ... $<a,p_n,b>$. Labeled digraphs thus correspond to the familiar *semantic net* construct of AI.

Predicate Representations

Given the problem of representing a unary predicate P(x) over a finite domain D of fixed size n, an obvious and familiar solution is to use boolean vectors, a.k.a. bit strips: for any d_i in D, P(d_i) is true (false) iff the i'th component of the vector representing P is a 1 (0). Boolean operations such as AND, OR, and NOT on predicates over D are then representable by the corresponding operations over bit strips, which are efficient on most computers. Similarly, binary predicates Q(x,y) over D x D can be efficiently represented by *adjacency matrices*, i.e. n x n matrices whose ij element is 1 if Q(d_i,d_j) is true, else 0.

Symbolic Adjacency Matrices

Boolean adjacency matrices can in principle represent labeled digraphs: a separate matrix is assigned to each label, and represents the subgraph of nodes connected by edges bearing that label. In practice this representation can become unwieldy. The number of different labels may be large, resulting in proliferation of adjacency matrices. Moreover, queries such as "is there any path (regardless of labels) from node a to node b?" require that the matrices

for all labels be ORed together. An answer to the follow-up query "what are these paths?" is even more difficult to generate from this representation. Such considerations motivate the adoption of symbolic adjacency matrices (SAMs) as representation for labeled digraphs. Element ij of a SAM is P iff the arrow from d_i to dj in the semantic net has label P, else NIL.

Warshall's Algorithm

Let G be a digraph; then the transitive closure G* of G is a digraph containing an edge <a,b> iff G contains a *path* (of length 0 or greater) from a to b. Warshall's Algorithm (see [6]) is an efficient method for computing G*, given an adjacency matrix representing the G. Intuitively, the algorithm scans the matrix top to bottom, left to right. If a 1 is encountered, say in row i, column j, then row i is replaced by row i OR row j, and the scan continues from position ij.

A straightforward extension, described in [4], of Warshall's Algorithm to symbolic adjacency matrices produces a matrix, termed the *path symbolic adjacency matrix* (PSAM), whose ij entry contains the set of all paths from node i to node j. It the generation of the PSAM matrix that makes the quantification over paths feasible.

REFERENCES

[1] Abbott, K., *Robust Fault Diagnosis of Physical Systems in Operation,* Ph.D. Dissertation, Computer Science Department, Rutgers University, New Brunswick, NJ, May 1990.

[2] Charniak, E., et al., *Artificial Intelligence Programming*, 2nd ed., Lawrence Earlbaum Associates, 1987.

[3] Clocksin, W., and C. Mellish, *Programming in Prolog*, Springer-Verlag, 1981.

[4], Feyock & S. Karamouzis, LIMAP, Technical Report, Computer Science Dept., College of William & Mary, in preparation.

[5] Feyock, S., and Dalu Li, *Simulation-Based Reasoning About the Physical Propagation of Fault Effects,* Proc. of the 1990 Goddard Conference on Space Applications of AI, May, 1990.

[6] Horowitz, E., & S. Sahni, *Fundamentals of Data structures*, Computer Science Press, 1976

[7] Schutte, P., *Real-time Fault Monitoring for Aircraft Applications using Qualitative Simulation and Expert Systems,* in Proc. of the AIAA Computers in Aerospace VII Conference, Monterey, CA, October 1989.

[8] Steele, Guy, *Common LISP: the Language*, Digital Press, Bedford, MA, 1984.